面向众核结构的并行 Comba 乘法研究 *

黄皓冉，徐江峰

(郑州大学 信息工程学院，郑州 450001)

摘　要：为发挥众核处理器性能优势及求解更大规模问题，针对大整数乘法在众核处理器上的并行化进行研究。在对笔算乘法和 Comba 乘法并行性进行分析的基础上，针对 Comba 乘法并行化时面临的负载均衡问题并提出了多种解决方法；然后针对 SW26010 的结构特征，选择借鉴笔算乘法改进的 Comba 乘法，并且实现过程使用了向量化、寄存器通信等优化方法。测试结果说明改进后的并行 Comba 算法具有较好的并行性，能够很好地利用 SW26010 众核处理器的性能优势。

关键词：大整数乘法；Comba 乘法；众核处理器；并行化；负载均衡

中图分类号：TP301.4　　doi: 10.3969/j.issn.1001-3695.2018.03.0159

Research on parallel Comba multiplication for many core processors

Huang Haoran, Xu Jiangfeng

(*College of Information Engineering*, *Zhengzhou University*, *Zhengzhou* 450001, *China*)

Abstract: In order to exploit the performance advantages of many core processors and solve more large-scale problems, this paper studied the parallelization of large integer multiplication on many core processors. Based on the parallel analysis of Written multiplication and Comba multiplication, this paper proposed a variety of solutions to solve the problem of load balancing when Comba multiplication is parallelized. Then according to the structural characteristics of SW26010, this paper selected the improved Comba multiplication based on Written multiplication. And the implementation process used some optimization methods such as vectorization, register communication and so on. The test results show that the improved Comba algorithm has better parallelism and can make good use of the performance advantages of the SW26010 many core processor.

Key words: large number multiplication; Comba algorithm; many core processor; parallelization; load balancing

0　引言

对算数运算的研究一直是人类科学发展过程中非常重要的方向，很多科学问题的解决依赖于准确、快速的算术运算。在计算机上，算数运算是实现各种应用的基础，研究其高效算法对利用计算机解决科学问题有着重要意义。在加减乘除四则基本算数运算中，乘法运算，尤其是大整数乘法运算，在密码系统[1]、大规模科学计算中有着广泛的应用。随着科技的发展，需要求解的问题的规模越来越大，精度要求越来越高，对大整数乘法的性能要求也越来越高。长久以来，人们对大整数乘法进行了大量研究，成果众多，形成了多种大整数乘法计算方法。主要的乘法算法大致可以分基本乘法、分治乘法、FFT（fast Fourier transform）乘法和数论乘法等几类。

基本算法主要是传统的笔算算法以及在此基础上的改进算法。笔算乘法中，一个乘数的每一位需要乘以另一个乘数的每一位，因此，两个 n 位整数相乘，笔算乘法的时间复杂度为 $O(n^2)$。1990 年，Comba 在笔算乘法的基础上提出了一种改进算法[2]，虽然执行乘法的次数与笔算乘法相同，但 Comba 算法以列为单位计算，将笔算乘法中处理进位的次数由 n^2 次降为 2n 次，Comba 算法普遍用于 PC 加密系统。基本算法最为简单，具有容易理解，并且易于计算机程序实现的特点。

分治乘法、FFT 乘法和数论乘法等快速算法都将乘法运算的复杂度降低到了 $O(n^2)$ 以下。分治是以一种解决大规模问题的重要方法，Karatsuba[3]使用二分递归的方法将大整数乘法的时间复杂度降低到了 $O(n^{1.585})$，该算法实现相对简单，额外开销较少，在 RSA 等公钥加密系统中有较多使用。Toom 发展了这种分治递归的思想，提出将大整数分为 k 部分（k≥2）进行递归，k 随乘法规模而变化，后来 Cook 利用 Toom 的思想加快乘法的计算机程序，称为 Toom-Cook 乘法算法[4]。该算法在 k 值较大时变得非常复杂，在实际应用中较少采用，常用的 k 值一般在 6 以下。当 k=3 时，就是最常用的 Toom-Cook 3-Way 算法，其时间复杂度为 $O(n^{1.465})$。大整数乘法利用 FFT 计算过程

收稿日期：2018-03-14；**修回日期：**2018-05-07　**基金项目：**国家“863”计划资助项目（2014AA01A300）

作者简介：黄皓冉（1991-），男，硕士研究生，主要研究方向为高性能计算（naduo0601@outlook.com）；徐江峰（1965-），男，教授，博士，主要研究方向为数据加密、数字水印.

中多项式在系数表示与点值表示之间快速转换，能够将时间复杂度能降低到 $O(n\log n\log\log n)$[5]。但是 FFT 乘法的缺点是可能产生精度误差，利用数论变换（Number Theoretic Transforms，NTT）中的原根代替 FFT 中的单位负根实现的数论乘法则能克服这一问题[6]。此外，2007 年，Fürer 提出了一种理论时间复杂度为 $O(n\log n2^{O(\log^* n)})$ 的乘法算法[7]，是目前最快的大整数乘法算法，此后又有大量对该算法的改进算法[8][9]，该算法太过复杂，目前仅在超大整数运算中有所应用。

这些算法使用了不同的计算策略，并各具优势。在实际应用中，无法通过某一方法解决所有问题，根据各个算法的优势，在不同的数据规模下使用不同的算法是大整数运算库常用的策略，比如性能非常优秀的 GMP 库[10]使用了基本算法，FFT 算法，分治算法等多个算法。笔算乘法和 Comba 乘法等基本乘法算法较其他快速算法虽然复杂度高，但基本乘法实现简单，程序流程简单，在数据规模较小时有很大优势，而且基本乘法通常作为分治算法迭代到较小规模时调用的底层算法。

目前，在功耗和散热等问题的限制下，CPU 主频的提升达到上限，单个 CPU 的计算能力已经很难继续提升。为了继续维持摩尔定律，计算机体系结构开始向多核、众核结构发展，多核处理器已经十分普及，众核处理器也开始在高性能计算领域普遍使用[11]。例如，目前在超级计算机 TOP500 榜单中排名第一的我国神威太湖之光计算机使用的具有自主知识产权的 SW26010 众核处理器[12]，单芯片核数达到了 260 核。研究如何在新型众核结构上高效实现并行的大整数运算，特别是大整数乘法运算，对于发挥众核处理器性能优势，求解更大规模问题具有重要意义。但目前对并行大整数乘法算法的探讨和研究还相对较少，仅有少量研究如何在多核/众核平台并行实现现有乘法算法的工作。如中科院的蒋丽娟等人开展了 Comba 和 Karatsuba 乘法的多核并行化的研究工作[13]；许亮基于 CUDA 实现了 FFT 乘法[14]，赵明祥在 MIC 加速器上实现了一个并行的 Karatsuba 算法[15]。但对算法并行性的分析，以及如何提高算法并行性，并解决并行化过程中的负载均衡等问题的研究较少。

本文的研究目的是找到能够在众核结构上高效并行的基本乘法算法，为众核结构上构建完整的大整数乘法算法奠定基础。本文在研究分析了笔算乘法、Comba 算法并行性的基础上，提出了多种实现高效并行 Comba 乘法的改进方法，并在 SW 众核处理器上进行了实现，实现过程采用了片上寄存器通信、向量化等优化方法。

1 基本乘法分析

1.1 笔算乘法分析

笔算乘法是最早的乘法运算方法，具有易于理解、实现简单的特点。笔算乘法中一个乘数的每一位需要与另一个乘数的每一位相乘，并将这些乘积按进位比例累加到一起。下面给出笔算乘法的具体算法描述。

笔算乘法：给定两个 b 进制整数 $(u_{m-1}\ldots u_1u_0)_b$ 和 $(v_{n-1}\ldots v_1v_0)_b$，它们的乘积记为 $(w_{m+n-1}\ldots w_1w_0)_b$。

M1. [初始化] 将 $w_{m+n-1}, w_{m+n-2}, \ldots, w_0$ 赋初值为 0，置 $i\leftarrow 0$。

M2. 置 $j\leftarrow 0$，$k\leftarrow 0$。

M3. [部分乘积累加] 置 $t\leftarrow u_i\times v_j+w_{i+j}+k$，然后置 $w_{i+j}\leftarrow t\%b$，$k\leftarrow t/b$

M4 [对循环 j] j++，如果 j<n，则返回 M3，否则令 $w_{i+j}\leftarrow k$

M5 [对循环 i] i++，如果 i<m，则返回 M2，否则算法结束

根据上述笔算算法，可以得到用 C 语言实现笔算乘法的核心是一个 2 层嵌套循环：

```
for (i = 0; i < m; i++)
{
    k = 0;        //表示进位
    for(j = 0; j < n; j++)
    {
        t = u[i] * v[j] + w[i+j] + k;
        w[i+j] = t%10;
        k = t/b;
    }
    w[i+j] = k;
}
```

分析笔算乘法发现，用笔算乘法计算两个 m 位和 n 位数的乘积，至少有需要进行 mn 次乘法，以及 mn 次进位处理，所以，笔算乘法计算两个 m 位和 n 位数乘积的时间复杂度为 o(mn)。特别地，当两个乘数位数相同时，复杂度为 $o(n^2)$。

笔算乘法计算过程中，每一次相乘都需要向上传递进位，进位处理使得笔算乘法每一步的计算都依赖于上一步的计算结果，只能顺序执行。上述 C 语言实现的笔算乘法的核心循环中，j 循环的每次迭代都依赖于上一次迭代计算的进位 k，存在循环携带的依赖；i 循环的每次迭代也需要上一次迭代计算的进位 w[i+j]，同样存在循环携带的依赖。因此，笔算乘法过程只能顺序执行，整个算法很难并行执行，不具有好的并行性。

1.2 Comba 乘法分析

1990 年，Comba 在笔算乘法算法基础上提出了 Comba 乘法。Comba 乘法法的计算过程于乘法类似，同样需要一个乘数的每一位与另一个乘数的每一位相乘，不同的是 Comba 算法以列为单位进行计算，首先合并整列，然后再处理整列的进位，减少了笔算乘法中处理进位的次数，算法效率有较大的提升。下面给出 Comba 乘法的具体算法描述。

Comba 乘法：给定两个 b 进制整数 $(u_{m-1}\ldots u_1u_0)_b$ 和 $(v_{n-1}\ldots v_1v_0)_b$，它们的乘积记为 $(w_{m+n-1}\ldots w_1w_0)_b$。

M1. [初始化] 将 $w_{m+n-1}, w_{m+n-2}, \ldots, w_0$ 赋初值为 0，置 $i\leftarrow 0$，$k\leftarrow 0$。

M2. [计算 u，v 位置及位数] 置 $y\leftarrow \min(i,n-1)$，$x\leftarrow i-y$，$s\leftarrow \min(m-x,y+1)$，$t\leftarrow 0$。

M3. [列乘积] 计算 $t\leftarrow t+u_x * v_y$，并 x++，y--

M4 [对循环 j] j++，如果 j<s，则返回 M3，否则计算 t←t+k，k←t/b，w[i]←(t)%b

M5 [对循环 i] i++，如果 i<m+n，则返回 M2，否则算法结束

根据上述算法，可以得到用 C 语言实现 Comba 乘法的核心如下：

```
for(i = 0, k=0; i < m+n; i++)
{
    t = 0;
    y = min(i,n-1);
    x = i -y;
    s = min(m-x,y+1);
    for(j = 0; j < s; j++)
    {
        t += u[x++] * v[y--];
    }
    t += k;
    w[i] = t%b;
    k = t/b;
}
```

分析 Comba 乘法可知，用 Comba 乘法计算两个 m 位和 n 位数的乘积，至少有需要进行 mn 次乘法，与笔算乘法相同。但只需要处理 m+n 次进位，将笔算乘法处理进位的次数由 mn 降为线性。Comba 乘法计算两个 m 位和 n 位数乘积的时间复杂度同样是 o(mn)。特别地，当两个乘数位数相同时，复杂度为 $o(n^2)$。

在 Comba 乘法的计算过程中，以列为单位进行计算，每一列的部分乘积合并过程中无需传递进位，合并完一列之后再计算整列的进位并向高位传递，得到最终结果。在上述 Comba 算法的经典实现中，外层的 i 循环用来遍历每一列的计算，i 循环存在进位 k 引起的循环携带依赖，使得 i 循环的计算只能顺序执行。内层的 j 循环用来完成第 i 列的合并，由于本身是一个规约求和的过程，而且循环的次数是随 i 循环变化的，变化范围从 1 到 n，也能高效并行。所以，Comba 算法并行性同样不好。

2 Comba 乘法改进

笔算乘法和 Comba 乘法都不具有好的并行性，很难在众核处理器上进行高效的并行使实现。但分析 Comba 乘法的原理可知，在 Comba 乘法计算过程中，每列的计算本质上是相互独立的，能够并行执行，所以本文选择 Comba 乘法进行改进，寻求在众核处理器上并行实现的方法。

只需将 Comba 算法的实现过程稍作改进，将整个计算过程分为两步，第一步计算各列，第二步统一处理进位，如下 C 代码段所示：

```
//第一步：合并各列
for(i = 0; i < m+n-1; i++)
{
    t = 0;
    y = min(i,n-1);
    x = i -y;
    s = min(m-x,y+1);
    for(j = 0; j < s; j++)
    {
        w[i] += u[x++] * v[y--];
    }
}
//第二步：逐列处理进位
for(i = 0,k = 0; i < m+n; i++)
{
    t = w[i] + k ;
    w[i] = t%b;
    k = t/b;
}
```

这样就能使第一步中每列的计算过程相互独立，可以完全并行，提升 Comba 算法的并行性。但是第二步从低位到高位依次处理进位的过程需要顺序执行，无法并行。

第一步中各列的计算虽然可以完全并行，但该实现方法在并行时存在一个严重缺陷，即各列的计算需要进行的单精度乘法的次数不同。这使得并行计算各列的值时，各列的计算量存在很大差异，会导致严重的复杂不均衡现象。要解决负载不均衡的问题有两种思路：一种是在任务划分的时候考虑各列计算量不同的情况，寻找负载更加均衡的任务划分方式；另一种是对 Comba 算法继续进行改进，寻求更优的实现方法。下面对这两种思路分别进行分析。

2.1 任务划分策略

图 1 给出了 Comba 乘法计算过程的示意图。除最高位的可能的进位处理外，共有 m+n-1 列需要计算，前 n-1 列和后 n-1 列的计算中单精度乘法运算的次数都不足 n 个，而且是随着列均匀变化；只有中间的 m-n+1 列的计算中单精度乘法运算的次数为 n。于是在进行任务划分时，对于前 n-1 列，可将第 1 列与第 m+1 列作为一个整体划分到一个任务中，第 2 列和第 m+2 列作为一个整体，……，以此类推。这样共有 m 个计算任务需要划分，且每个任务中单精度乘法的次数相同。

任务划分的方法能够较好的解决负载不均衡的问题，但是存在着一些不足。比如用 OpenMP 和 OpenACC 等较高层的基于编译指示的编程模型，难以实现如此精细的任务划分；用 Pthread 和 MPI 等需要手工进行任务划分的编程模型，虽然可以实现任务的细粒度控制，但实现过程很繁琐，而且每个任务计算的列数不同会导致每个进程或线程的通信量不同，通信代码很难编写。

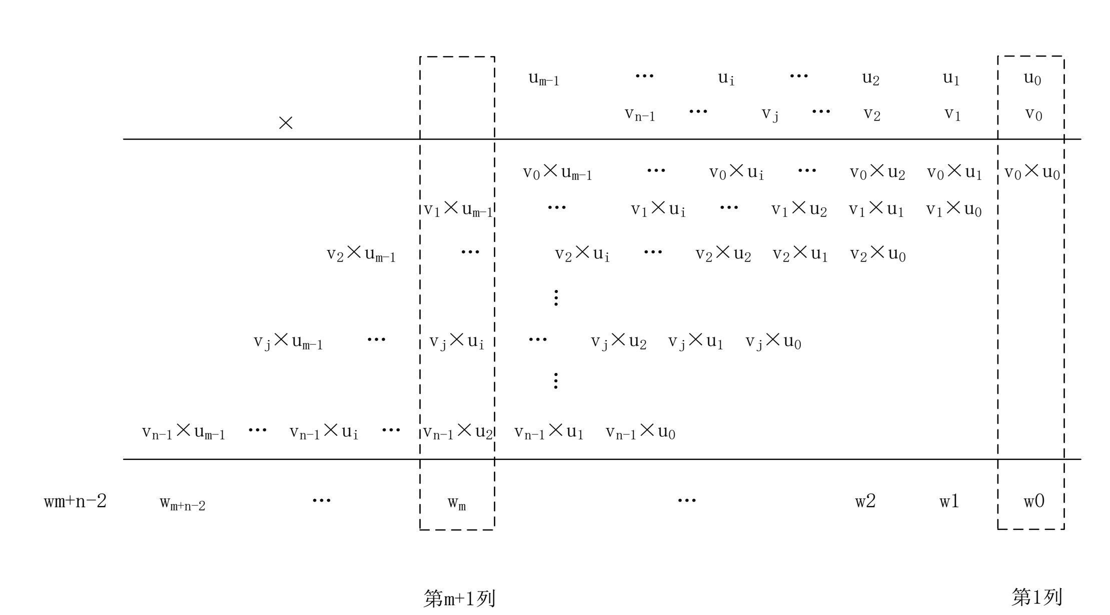


图 1　Comba 乘法计算过程示意图

2.2　改进的 Comba 算法

由于任务划分的方法存在一定不足，本文提出了两种对 Comba 实现进行改进的方法。

1）调整计算顺序

借鉴上述任务划分时的策略，在实现 Comba 算法时，即把第 i 列和第 i+m 列的计算放到一次循环迭代中。实现代码如下：

```
//第一步：合并各列
for(i = 0; i < m; i++)
{
    y1 = min(i,n-1);
    y2 = y1+1;
    x1 = i -y1;
    x2 = i+m-y2;
    s = min(n,i+1);
    for(j = 0; j < s; j++)
    {
        w[i] += u[x1++] * v[y1--];
    }
    for(j = 0; j < n+1-s; j++)
    {
        w[i+m] += u[x2--] * v[y2++];
    }
}
```

该方法继承了上述任务划分方法的优点，同时克服了任务划分的方法用 OpenMP 和 OpenACC 等编程模型无法实现的缺点，使得并行程序的设计人员能够更多的关注算法的并行实现上，而不去分析算法本身存在的不足。

2）借鉴笔算乘法

结合笔算乘法与 Comba 乘法的特点，在计算顺序上借鉴笔算乘法。改进后的算法不再按顺序计算结果的各个列，而是顺序遍历 u 和 v 两个乘数的各个位。由乘法的计算法则可知，对于 u_i 和 v_j 两个位数，其乘积是结果 w_{i+j} 的一部分，即有 $w_r = \sum_{i+j=r} u_i * v_j$，遍历完两个乘数后即可累加得到各列的结果。这里与笔算乘法的区别是部分乘积累加的过程不再处理进位，进位是在得到了各列的值之后统一处理。

于是，可以将 Comba 算法的第一步可以改进为如下实现：

```
//第一步：计算得到各列
for(i = 0; i < m; i++ )
    for(j = 0; j < n; j++)
        w[i+j] += u[i] * v[j];
```

在该实现中，Comba 乘法的第一步简化为一个 2 层完美嵌套循环，且 2 层的循环执行次数都是固定的，其迭代空间是一个矩形。该实现程序控制逻辑简单，而且对实现优化有很多方便之处，比如数据预取、向量化等都便于在该循环上进行。但该方法的不足之处是，在对 i 循环划分之后进行并行执行时，每个线程或进程上得到的都是每一列的部分值，还需要要把每个线程或进程上的 w 数组通信到一个线程或进程上进行一次累加。该方法在通信代价较大的机器上不太适用，但在共享存储结构上，由于进程或线程之间进行数据通信的代价很小，该方法能够取得很好的效果。

3　并行 Comba 算法在 SW 众核处理器上的实现

3.1　SW 众核处理器介绍

本文的目标是在国产 SW26010 众核处理器上实现高效的基本乘法算法。SW26010 是太湖之光计算机系统实现高计算速度和低功耗的核心器件。SW26010 是面向高性能计算领域开发的处理器，其结构如图 2 所示。

芯片主要 4 个核组、片上互联网络和系统接口等组成。核组内采用异构众核结构，包括 1 个主核和 1 个由 64 个从核构成的加速核心阵列。主核是功能完整的 64 位 RISC 核心，支持 32/64 位整数运算、单/双精度浮点运算和原子操作，可以高效

处理程序的串行段，并完成计算资源和功耗的管理。从核也是 64 位 RISC 结构，其指令集在主核指令集的基础上进行了精简，并针对从核结构特征增加了寄存器通信、行列同步等特殊指令。

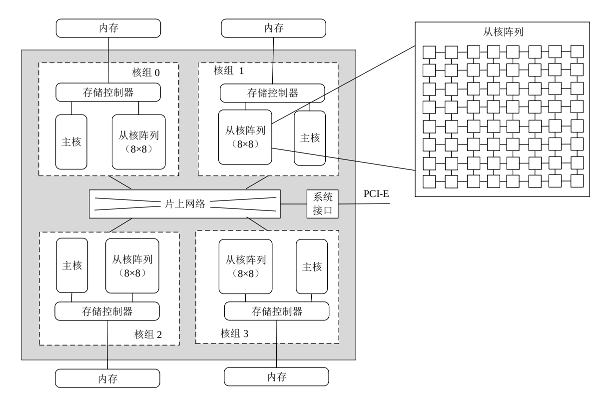


图 2　SW26010 处理器结构图

3.2　并行 Comba 乘法的实现

SW26010 单个处理器包含四个核组，核组是神威太湖之光计算机系统上进行资源分配的基本单位。本文的目标是在单个核组上实现快速的并行大整数乘法。SW26010 上实现单个核组内主核和从核之间的异构并行有两种编程方法：一种是使用基于编译指示的 OpenACC*编程模型[12]；另一种是使用更为底层的 API 接口 Athread。OpenACC*基于编译指示实现，编程简单，但程序运行效率较低。而层次更低的 Athread 能够对任务划分以及数据传输、同步等进行更加精细的控制，虽然编程复杂，但能够得到更好的程序效率。为追求更高效率，本文选择用 Athread 进行实现。

在 SW26010 上实现并行 Comba 时，根据 SW26010 处理器的特性，本文选择使用借鉴笔算乘法改进的 Comba 算法。选择的原因主要是该方法实现简单，能够节省很多程序逻辑控制的开销，而且 SW26010 支持从核间的寄存器通信，能够高效的实现从核之间的数据通信。Comba 算法主要用于位数较小的乘法，或者在分治算法把乘数递归到较小规模时调用，所以实际通信的数据量不会很大，用寄存器通信能够实现快速的通信。

在 SW26010 上实现 Comba 算法时，从核阵列并行地完成 Comba 乘法第一步各列的计算，主核完成第二步的进位处理。实现的基本思路如下：

a)对 i 循环进行划分，每个从核执行 m/64 次迭代。为了简化分析过程，这里假设 m 能够整除 64，对于实际中不能整除的情况，可以在高位补 0。这样，每个从核上都会计算出乘积 w 数组各个元素上的一部分值。在这步的计算过程中，内层的 j 循环没有依赖，而且访存也是连续的，能够很方便的进行向量化。

b)将各个从核上计算得到的 w 数组进行规约。一个核组的 64 个从核按照 8×8 的网络结构进行连接，支持行和列上的广播通信。在将结果进行规约时，首先利用从核之间的片上通信，将每一行 8 个从核上的 w 数组进行规约，将每一行的数据规约到编号最小的从核上（即所有的结果位于第一列）；然后在第一列上进行一次规约，将结果规约到一个从核上；最后通过 DMA 将规约得到的 w 数组从从核局存传输到主存。

c)主核顺序完成第二步各列进位的处理。进位处理中需要用到求余和除法运算，这两个运算比较耗时，如果大整数表示采用的进制为 2 的幂，则可以将求余和除法运算用位运算快速实现。

4　测试与分析

本节对本文改进和实现的并行 Comba 乘法的性能进行测试与分析。实验平台为神威太湖之光计算机系统，其计算节点为 SW26010 众核处理器，操作系统为 Raise Linux，配备有完整的编译和性能分析工具链。本文选择了最直观和在科学计算中最为常见的 10 进制表示方法进行算法实现和测试。

Comba 乘法通常在数据规模较小时调用，所以本文测 Comba 乘法使用了较小规模的数据。测试使用了 10 组随机生产的数据，乘数位数在百位和千位两个量级。Comba 乘法并行和串行所用的时钟周期对比如表 1 所示，加速比的计算方法为串行时钟周期数除以并行时钟周期数。并行版本为本文改进后在 SW26010 上实现的方法，串行版本则采用的经典的 Comba 乘法实现方法。这里并行执行是指在一个核组上利用主核控制从核阵列进行加速计算，串行执行则是在单个主核上运行。

表 1　Comba 乘法众核测试结果

乘数 1 位数	乘数 2 位数	并行时钟周期数	串行时钟周期数	加速比
284	351	10488	325059	30.99
586	598	20015	920980	46.01
664	2681	91656	4678102	51.04
821	5884	190685	11816065	61.97
1086	2624	87930	7200938	81.89
2654	3521	255391	21955324	85.97
4892	6834	1304104	89852601	68.90
6534	6987	2312736	129658834	56.06
8364	7654	3857763	185600336	48.11
9353	8632	6967634	237715560	34.12

从测试数据中可以看出，本文实现的并行 Comba 程序能够显著提升 Comba 乘法的性能，在 SW26010 众核处理器上加速比平均达到 57.75。在乘数位数在 1000-4000 位左右时程序获得的加速效果最好，这是因为规模较小时，从核启动等并行启动的固定开销所占比重较大；而数据规模增大时从核片上的通信量快速增大，导致通信性能下降。

虽然针对大整数乘法的并行优化已有较多的研究工作，但这些研究工作面向的平台不同，采用的表示方法不同，很难直接对比不同工作的优化效果。例如，中科院的蒋丽娟在 8 核处理器上采用 OpenMP+SIMD 实现的并行 Comba 乘法平均获得 5.63 的加速比[13]；乌克兰的 Vladislav Kovtun 和 Andrew Okhrimenko 在对 Comba 算法进行改进后，4 线程情况下使用 section 策略和 for 策略实现并行时可分别获得 1.5 倍及 2 倍加速[16]。与蒋丽娟使用 8 核时获得 5.63 加速比以及 Vladislav Kovtun 和 Andrew Okhrimenko 在 4 线程下获得 1.5-2 倍加速相比，本文程序在 65 核心（1 个主核+64 个从核）的 SW26010 单个核组上，采用并行化、向量化等优化方法后平均获得 57.75 加速，在并行规模扩大的情况下，程序并行效率仍有优势。

测试结果一方面说明本文改进后实现的并行 Comba 算法本身有较好的并行性，而且代码实现上也充分发挥了 SW26010 众核处理器的性能优势；另一方面也说明国产 SW26010 众核处理器在处理大整数运算方面也能发挥好的效果。

5 结束语

基于计算机体系结构的发展以及现实中对高效大整数乘法需求的提升，本文对众核处理上的大整数基本乘法进行了研究。在对笔算乘法和 Comba 乘法并行性进行分析的基础上，本文对 Comba 乘法进行了详细研究，分析了 Comba 乘法并行化时面临的负载均衡问题并提出了多种解决方法。最后针对国产众核处理器 SW26010 的结构特征，选择了借鉴笔算乘法改进的 Comba 乘法，并且进行了优化实现。在并行 Comba 乘法的基础上，后续我们会对其他乘法方法的并行化继续进行研究，在此基础上构建完整的并行大整数乘法系统。

参考文献：

[1] Wang Wei, Huang Xinming, Emmart N, *et al.* VLSI design of a large-number multiplier for fully homomorphic encryption [J]. IEEE Trans on Very Large Scale Integration Systems, 2014, 22 (9): 1879-1887.

[2] Moore C, O'Neill M, Hanley N, *et al.* Accelerating integer-based fully homomorphic encryption using Comba multiplication [C]// Signal Processing Systems. 2014: 1-6.

[3] Karatsuba A. The complexity of computations [J]. Proceedings of the Steklov Institute of Mathematics, 1995, 211: 169-183.

[4] Lee C Y, Meher P K, Lee W Y. Subquadratic space complexity digit-serial multiplier over binary extension fields using Toom-Cook algorithm [C]// Proc of International Symposium on Integrated Circuits. 2015: 176-179.

[5] Schönhage A. Fast multiplication of polynomials over fields of characteristic 2 [J]. Acta Informatica, 1977, 7 (4): 395-398.

[6] Feng Xiang, Li Shuguo. Design of an area-effcient million-bit integer multiplier using double modulus NTT [J]. IEEE Trans on Very Large Scale Integration Systems, 2017, 25 (9): 2658-2662.

[7] Fürer M. Faster integer multiplication [J]. SIAM Journal on Computing, 2009, 39 (3): 979-1005.

[8] Covanov S, Thomé E. Fast arithmetic for faster integer multiplication [J]. Computer Science, 2015.

[9] Covanov S, Thomé E. Fast integer multiplication using generalized Fermat primes [J]. Computer Science, 2016.

[10] The GNU multiple precision arithmetic library [S/OL]. (2018-03-12) https://gmplib. org/

[11] 廖湘科，肖侬. 新型高性能计算系统与技术 [J]. 中国科学：信息科学, 2016, 46 (9): 1175.

[12] Fu Haoheng, Liao Junfeng, Yang Jinzhe, *et al.* The sunway TaihuLight supercomputer: system and applications [J]. Sciece China Information Sciences, 2016, 59: 1-16.

[13] 蒋丽娟，刘芳芳，赵玉文，等. 大整数 Comba 和 Karatsuba 乘法的多核并行化研究 [J]. 计算机系统应用, 2016, 25 (11): 232-236. (Jiang Lijuan, Liu Fangfang, Zhao Yuwen, *et al.* Multi-core parallel of large integer multiplication Comba and karatsuba algorithms [J]. Computer Systems & Applications. 2016, 25 (11): 232-236)

[14] 许亮，王震. 基于 CUDA 的快速大整数乘法 [J]. 计算机工程与应用, 2013, 49 (16): 221-224. (Xu Liang, Wang Zhen. Fast large integer multiplication based on CUDA [J]. Computer Engineering and Applications, 2013, 49 (16): 221-224.)

[15] 赵明祥. 基于 MIC 加速部件的长整数基础计算 [D]. 广州：华南理工大学, 2016. (Zhao Mingxiang. The basic calculation of long integer on MIC acceleration components [D]. Guangzhou: South China University of Technology, 2016.)

[16] Kovtun V, Okhrimenko A. Approaches for the performance increasing of software implementation of integer multiplication in prime fields [J]. Iacr Cryptology Eprint Archive, 2012.